Diseño de una micro-arquitectura para el monitoreo de un Sistema de Gestión de Órdenes omnicanal centralizando transacciones en un motor de búsqueda distribuido

Andrés Echeverry
Departamento de Ingeniería de Sistemas
Universidad del Norte
Barranquilla, Colombia
aeecheverry@uninorte.edu.co

Jefferson Sierra
Departamento de Ingeniería de Sistemas
Universidad del Norte
Barranquilla, Colombia
sjefferson@uninorte.edu.co

Jair Arboleda
Departamento de Ingeniería de Sistemas
Universidad Del Norte
Barranquilla, Colombia
jaarboleda@uninorte.edu.co

Profesor asesor del proyecto
Wilson Nieto Bernal, PhD
Departamento de Ingeniería de Sistemas
Universidad del Norte
Barranquilla, Colombia
aeecheverry@uninorte.edu.co

***Abstract*—Innovative business models such as omnichannel retail create value for suppliers and consumers. Omnix is one of these models which was built on modern architectures such as microservices, due to, they support fast and continuous improvements of each functionality, however, the decoupling of components brings some challenges as; first, the monitoring of these systems becomes complex with the scalability and in complex applications eventually something can go wrong, because the number of microservices affects proportionally the management of components; and second, these models do not have a centralized management and monitoring system of transactions high and low level that allows them to address the failures or anomalies in real time. The Omnix Monitoring System or SIMO was born as a system oriented to support, notification and monitoring, which seeks to centralize and standardize the transactions of an omnicanal order management system, using a distributed search engine due to its high availability, resulting in improvements in response times and real-time monitoring.**

Keywords—e-commerce, monitoring, transactions processing, order management system, logs.

***Resumen*—Novedosos modelos de negocio como el retail omnicanal crean valor para proveedores y consumidores. Omnix es uno de estos modelos el cual se construyó sobre arquitecturas modernas como los microservicios, gracias a que estas soportan mejoras rápidas y continuas de cada funcionalidad, sin embargo, el desacoplamiento de componentes trae consigo algunos retos como; primero, el seguimiento de estos sistemas se complejiza con la escalabilidad y en las aplicaciones complejas eventualmente algo puede salir mal, ya que el número de microservicios afecta proporcionalmente la gestión de componentes y segundo, estos modelos no poseen un sistema de gestión y monitoreo de transacciones centralizado de alto y bajo nivel que les permita afrontar los fallos o anomalías en tiempo real. El Sistema de Monitoreo de Omnix o SIMO nace como un sistema orientado al soporte, notificación y monitoreo, que busca centralizar y estandarizar las transacciones de un sistema de gestión de órdenes omnicanal, usando un motor de búsqueda distribuido debido a su alta disponibilidad, dando como resultado mejoras en tiempos de respuesta y monitorización en tiempo real.**

Keywords—comercio electrónico, monitoreo, transacciones, sistemas de gestión de órdenes, registros.

I. Introducción

Los avances en la tecnología han permitido la creación de novedosos modelos de negocios que crean valor para los proveedores y consumidores; un ejemplo de esto es el retail omnicanal, siendo este una experiencia integrada que combina las ventajas de las tiendas físicas y el enriquecimiento y coherencia de la información de los e-commerce por la alta calidad del servicio que ofrecen los canales integrados y por la gran variedad de productos [1]. Este tipo de modelos se construyen sobre arquitecturas modernas como los microservicios, los cuales presentan un reto debido a que el monitoreo de sistemas basados en esta arquitectura complejiza la escalabilidad [2] en comparación con las arquitecturas tradicionales que poseen herramientas de gestión del rendimiento de una aplicación -APM- y soportan la medición de varias métricas en tiempo real [3]. Por lo tanto, la mayoría de sistemas de gestión de órdenes (**OMS,** por sus siglas en inglés) que están desplegados sobre una arquitectura de microservicios, como Omnix, no poseen un sistema de gestión de transacciones y monitoreo de alto y bajo nivel que les permita afrontar los fallos o anomalías del sistema en tiempo real [4]. Es decir, no posee un sistema centralizado de transacciones, o en su defecto cada microservicio se despliega sobre un servicio serverless independiente [5], por lo que se infiere que no existe una

estrategia para gestionar el alto volumen de transacciones, afectando los tiempos de solución de las incidencias. Por otra parte, los registros o transacciones no están estandarizados para hacer un filtrado efectivo de estas [6], razón por la cual cada consulta hace una búsqueda fullscan que afecta notablemente el rendimiento. En consecuencia a estas problemáticas se desarrolla un sistema orientado al soporte, notificación y monitoreo con el objetivo de centralizar y estandarizar las transacciones de un sistema de gestión de órdenes omnicanal, usando un motor de búsqueda distribuido (**Elasticsearch**) que, por su alta disponibilidad, da como resultado mejoras en tiempos de respuesta, monitorización en tiempo real, comprensión y análisis de información.

La metodología para el desarrollo del proyecto consiste en 6 fases, la primera es la fase de **investigación**; en la cual se realiza una búsqueda sistemática donde se extrae información que da forma a la arquitectura del sistema, estrategias de almacenaje de registros, estructuras de logs genéricas, registro de información en tiempo real de los microservicios para el análisis y diagnóstico del flujo de transacciones. La segunda es la fase de **análisis y comprensión del problema;** para entender la problemática de fondo se realiza diversas reuniones con los stakeholders de Omnix, asimismo se entrevista a los actores directos involucrados en el soporte, con el fin de particularizar y categorizar las causas del problema, esto para construir una mejor perspectiva del enfoque final del proyecto con el objetivo de poder realizar el estudio y planteamiento de requisitos acorde a las necesidades reales del cliente final. La tercera fase es el **modelamiento,** en esta, se indaga cómo se realiza la parte operativa y a partir de allí, se hace una revisión de documentos para realizar los modelos con representaciones gráficas de los procesos; esto para comprender el comportamiento funcional y obtener los requerimientos del sistema. La cuarta es la fase de **desarrollo e implementación,** para esta se identifica los componentes core del sistema y se jerarquiza en orden de prioridad para la puesta en marcha del desarrollo, se elaboran las actividades y se designa a un responsable. La quinta fase es el **despliegue**, en esta fase se realiza la documentación, integración y mitigación de cada componente de la arquitectura sobre el servicio en la nube correspondiente, el manual de usuario para el monitor web y en conjunción con la validación, se hacen reiteraciones continuas sobre los componentes. La sexta y última fase es la de **validación** en esta fase se verifica que cada uno de los componentes del sistema funcione de manera correcta mediante pruebas en todos los niveles de la arquitectura.

II. Descripción del problema

Cuando se hace referencia a sistemas basados en microservicios se piensa en componentes actualizables e intercambiables que pueden desplegarse o escalar independientemente según se requiera [2]. Gracias a que esta arquitectura soporta mejoras rápidas y continuas de cada funcionalidad, los proyectos construidos sobre ella se convierten en los principales candidatos para las metodologías ágiles, pero el desacoplamiento de componentes presenta algunos retos [8] como , por ejemplo, que las pruebas globales se complejizan precisamente a causa de la escalabilidad y en las aplicaciones complejas eventualmente algo puede salir mal, ya que el número de microservicios afecta proporcionalmente la gestión, mantenimiento e integración de los componentes [3][5]. Por ello, si no existe una forma de monitorear los servicios, se invisibiliza la administración y supervisión de errores, rendimiento y lógica de negocio aplicadas a los procesos, por lo que la identificación de estados degradados del sistema podría darse de forma tardía, generando a su vez un evento crítico, ya que estos estados degradados generalmente ocurren antes de fallos inminentes, y sin un seguimiento del comportamiento del sistema, los operadores no podrían tomar medidas preventivas antes de que ocurran fallas totales [16].

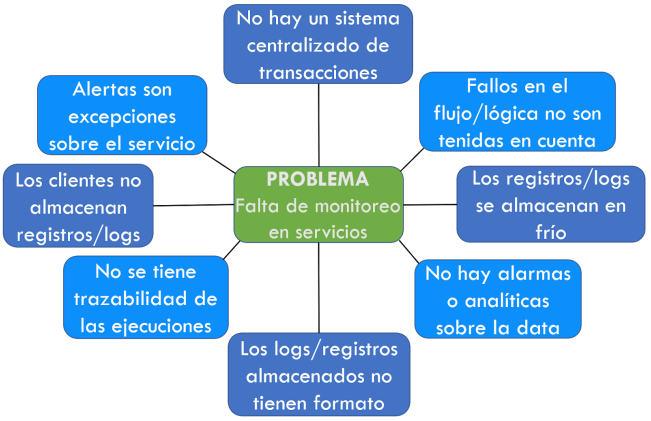


Fig. 1. Árbol del problema.

En la *figura 1*, se exponen las causas del problema. Por ejemplo, el almacenamiento adecuado de los logs o registros que pueden ser utilizados para distintos propósitos, como tener una versión más precisa de la realidad, hacer la reconstrucción de sucesos y la toma de decisiones tempranas [4], lo que permite anticiparse y optimizar el uso de los recursos disponibles. Es por esto que si no se almacenan adecuadamente, se producen dos problemas.

- El primero es una visibilidad limitada de errores para los equipos de desarrollo y/o sistemas, además de una metodología de trabajo no estandarizada [5], es decir, cada usuario aplicará su propia forma de trabajo basado en su experiencia.
- El segundo es que el acceso e información están descentralizados, lo que causa problemas para trabajar con los datos, y esto a su vez causa un incremento del tiempo de respuesta ante una incidencia, por lo que se verán afectados los acuerdos a nivel del servicio o **SLAs** (por sus siglas en inglés) [15].

De aquí parte la problemática principal de los *OMS,* ya que estos no cuentan con un sistema de gestión de transacciones centralizado para el manejo de altos volúmenes de datos, como consecuencia, la depuración de errores no es práctica o escalable [3][16], los registros almacenados no tienen importancia en el tiempo y/o las transacciones no están estandarizadas [7], es decir que no cuentan con un formato o arquetipo debido a que su estructura no es genérica, haciendo que estas no provean suficiente contexto, carezcan de información o contengan mucha información irrelevante, además de no tener una semántica clara en los mensajes [13]. Esto causa que, primero, las búsqueda sean fullscan, en otras palabras por cada consulta se hace una búsqueda sobre todo el dominio

de datos y segundo, no provea una trazabilidad del flujo de ejecución puesto que una transacción puede implicar llamadas a varios servicios. En consecuencia se ve afectado notablemente el rendimiento, la detección y prevención de errores, la cual tiene una dependencia del cliente y usuario final ya que son los primeros en percibir y reportar las incidencias, lo que causa un aumento en los costos operativos.

III. JUSTIFICACIÓN

El monitoreo se convierte en una parte crítica de la administración de los microservicios no sólo a bajo nivel sobre el performance de cada servicio sino a alto nivel con la lógica de negocios aplicada a un proceso que se distribuye entre varios servicios independientes. En consecuencia, el seguimiento de una aplicación a través de este mecanismo requiere una forma de correlacionar los servicios; [5] ya que los sistemas complejos, incluso los monolitos, pueden operar en un estado degradado que afecta el rendimiento por lo que se invierte mucho tiempo en identificar la causa del problema [7].

En **Omnix** se indexan las transacciones en *Elasticsearch,* de esta manera, se logra tener una visión real de la comunicación entre los servicios, los tiempos de respuesta y las respuestas de estos [15], ya que gracias a la centralización de las transacciones en un motor de búsqueda distribuido, se puede tener una trazabilidad del flujo de ejecución e incluso un seguimiento a través de los microservicios involucrados en la consulta. Este hecho es importante porque es el punto de partida que permite reducir o eliminar la dependencia del cliente y el usuario final, para tener una respuesta en tiempo real sobre las consultas que se hacen en los servicios. En otras palabras, ofrecer una reacción inmediata sobre las transacciones fallidas. El potencial de esta aplicación radica en que otorga la oportunidad de monitorear y prever incidentes no solo a nivel de servicios, sino que, gracias a que es un motor de búsqueda se pueden hacer consultas sofisticadas de forma granular. Además de eso, ahora las alertas y notificaciones son más enriquecedoras y precisas ya que contienen información suficiente de la ejecución, lo que permite dividirlas por niveles de prioridades, agrupar las alertas reiterativas y tener un contexto inmediato de los eventos que están sucediendo en los servicios [6]. Otro punto importante es el tiempo de respuesta de Elasticsearch, ya que a comparación de CloudWatch, cuyos tiempos varían entre segundos y decenas de minutos, los motores de búsquedas distribuidos ofrece un tiempo de respuesta de 1 segundo de media en las consultas, lo que a grandes rasgos es un ahorro potencialmente grande en términos de gastos operativos.

Por esto nace el interés de llevar a cabo esta propuesta como una solución real y a la vanguardia de los sistemas basados en transacciones.

IV. OBJETIVO GENERAL

Diseñar y desarrollar una micro-arquitectura como una solución de software orientada al soporte, notificación y monitoreo en tiempo real de las incidencias o anomalías ocurridas de un Sistema de Gestión de Órdenes omnicanal.

V. OBJETIVOS ESPECÍFICOS

- Elaborar la revisión sistemática de la literatura asociada con el desarrollo de software para la monitorización de un sistema de gestión de órdenes, al igual que la orquestación e implementación de microservicios en la nube.
- Diseñar la arquitectura de la solución basada en la implementación de microservicios desplegados en Amazon Web Service.
- Desarrollar el prototipo de la solución para el monitoreo en tiempo real del sistema y generación de reportes dinámicos a partir de la arquitectura de solución propuesta.
- Validación del prototipo logrado, conforme al SLA y a los requerimientos definidos.

VI. METODOLOGÍA

La metodología del proyecto es un elemento clave para alcanzar el objetivo propuesto, por ello es indispensable realizar una correcta planificación y gestión de recursos, ya que esto facilita la toma de decisiones, la minimización de riesgos y finalmente, la predicción de resultados. Nuestra metodología cuenta con 6 fases:

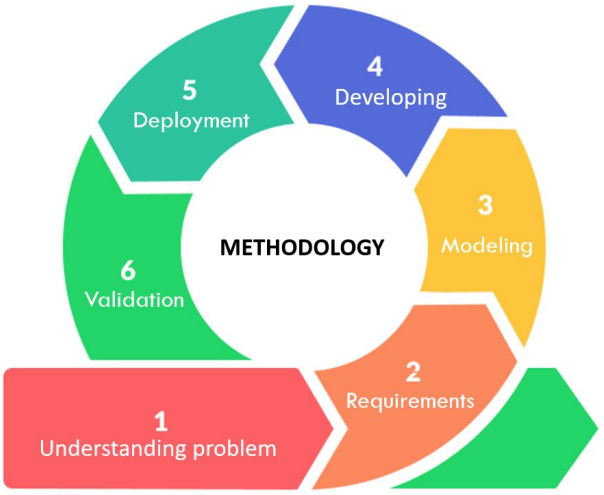


Fig. 2. Metodología.

Fase 1: Investigación
Para el diseño y prototipado del sistema se realiza una búsqueda sistemática de la literatura, se extraen artículos relevantes, se filtran y se hace una lectura en profundidad, con el objeto de definir la arquitectura, estrategias de almacenaje y monitoreo de información en tiempo real, estructuras genéricas de registros de los microservicios para el análisis y diagnóstico del flujo de transacciones.

Fase 2: Análisis y comprensión del problema
Se realiza diversas reuniones con los **stakeholders** de Omnix, asimismo se entrevista a los actores directos involucrados en el soporte, con el fin de particularizar y categorizar las causas del problema, esto para construir una mejor perspectiva del enfoque final del proyecto con el fin de poder realizar el estudio y planteamiento de los

requerimientos del sistema acorde a las necesidades reales del cliente final.

Fase 3: Modelamiento
Se parte de la premisa que el diseño de la arquitectura lógica de la solución debe ser capaz de resolver eficientemente los requerimientos funcionales, por ello primero se aborda cómo se está realizando actualmente la parte operativa, luego se hace una revisión de documentos y finalmente se realiza el modelado con representaciones gráficas de los procesos; con la finalidad de comprender el comportamiento funcional.

Fase 4: Desarrollo e implementación
Se identifican los dominios de **SIMO** para la segmentación, definición, documentación, e implementación de cada uno de los componentes **core** del sistema que contienen la lógica de negocio de los servicios los cuales se jerarquizan en orden de prioridad para la puesta en marcha del desarrollo, se elaboran las actividades y se designa a un responsable.

Fase 5: Despliegue
Se realiza la integración y mitigación de cada componente de la arquitectura sobre el servicio en la nube correspondiente, la documentación de **API** y en conjunto con la validación, se hacen iteraciones continuas sobre los componentes. Se usa **Amazon Web Service** para el despliegue de la arquitectura, **Elasticsearch** como motor de búsqueda, **MongoDB** como bases de datos de las entidades transversales, **Kibana** como framework para la creación de dashboards dinámicos y **Angular** para la construcción del Monitor Web.

Fase 6: Validación
Se verifica que cada uno de los componentes del sistema funcione de manera correcta mediante pruebas unitarias a nivel de servicio, con el uso de mocks de datos y callsFake de funciones para emular su respuestas. [10] Facilitando la búsqueda y corrección de errores en la etapa temprana del desarrollo.

VII. Arquitectura Lógica

La correcta organización de la estructura del proyecto evitará la duplicación de código, mejorará la estabilidad y potencialmente, ayudará a escalar los servicios.

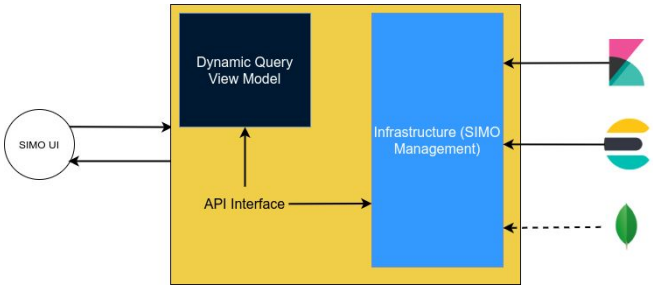


Fig. 3. Arquitectura de componentes.

La modularización de los diferentes niveles de la aplicación da al equipo de desarrollo la posibilidad de implementar y mejorar el sistema con mayor rapidez que el desarrollo de una base de código singular, ya que un nivel específico puede ser actualizado con el mínimo impacto en los otros niveles. También puede ayudar a mejorar la eficiencia del desarrollo al permitir que el equipo se centre en sus competencias básicas.

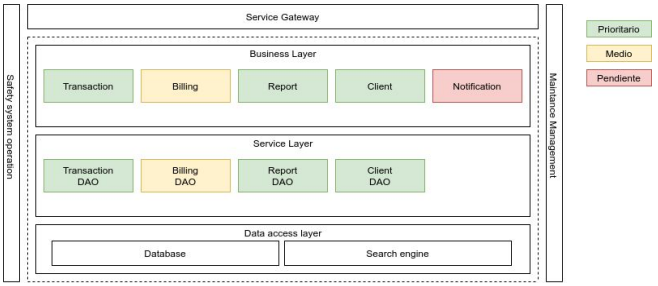


Fig. 4. Arquitectura lógica.

Por esto, para el core del sistema se utilizó el patrón de arquitectura '3-Layer Architecture' *ver fig. 4*, la cual nos permite separar la lógica en 3 niveles, todo bajo un modelo Cliente-Servidor. Los niveles que maneja este patrón son:

1. **Capa de presentación:** Es el primer nivel de la aplicación, se encarga de exponer los servicios, validar y enrutar las peticiones y construir la respuesta de los mismos.

2. **Capa de aplicación:** Es la capa que contiene el Business logic, la cual se encarga de ejecutar la función designada al servicio, que contiene la lógica del negocio del mismo.

3. **Capa de datos:** Es la capa de acceso a la información. En este nivel se realiza la conexión a la fuente de datos para realizar operaciones (Insert, Update, Delete) o consultas basadas en la información de entrada.

Este método elimina la dependencia de niveles, lo que permite trabajar en diferentes partes al tiempo, acelerando el proceso de desarrollo. Además, se genera un ahorro en tiempo y costo de mantenimiento, pues la implementación de nuevas características es más sencilla. Finalmente, se obtienen mejoras a nivel de seguridad al seguir esta arquitectura, pues la información no es accedida directamente sino a través del nivel de aplicación.

Uno de los mayores desafíos de los microservicios es definir los límites de los servicios individuales, ya que estos deben diseñarse alrededor de las funcionalidades de la empresa, para así garantizar un acoplamiento flexible y una alta cohesión funcional. El diseño basado en componentes proporciona una plataforma que puede ayudar en gran medida a diseñar bien los microservicios para garantizar que la arquitectura permanezca centrada en las funcionalidades del negocio.

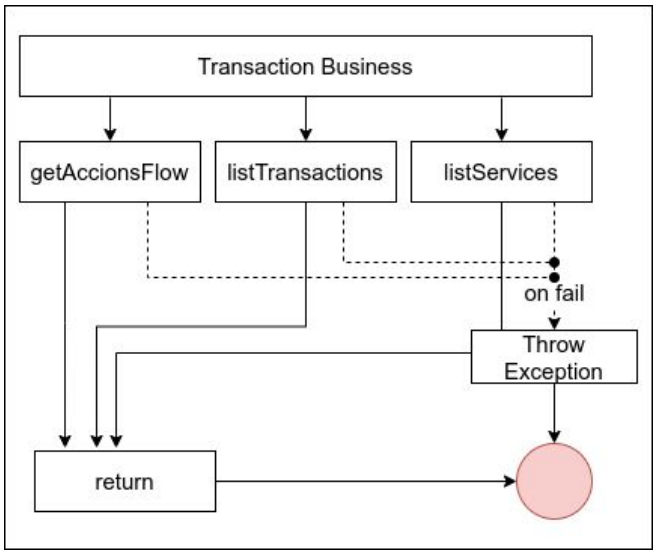


Fig. 5. Modelo de negocio de Transacciones.

Este componente contiene la lógica de negocio para la gestión y consulta intuitiva de transacciones, su función principal es dar acceso de forma centralizada a los eventos y flujos de ejecución que ocurren en los servicios de Omnix.

Consta de tres servicios principales.

- **Get Actions Flow:** Este servicio permite obtener el flujo de acciones sobre la orden y los shipping groups asociados.
- **List Transactions:** Este servicio permite la consulta de transacciones, usando diferentes criterios como: cadenas parciales, estado de una transacción, filtrado de atributos, rango de fechas y paginación.
- **List Services:** Este servicio permite obtener el listado de servicios de Omnix, usando los mismos criterios que la consulta de transacciones.

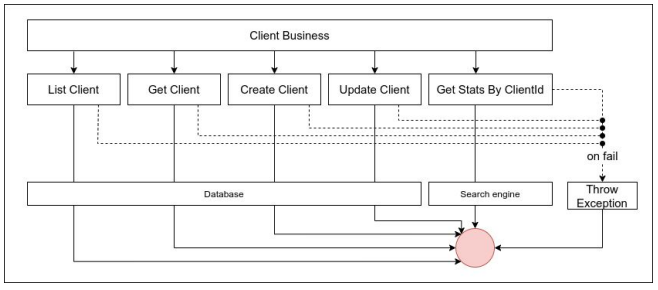


Fig. 6. Modelo de negocio del Cliente.

Este componente contiene la lógica de negocio para la gestión de clientes, su función principal es permitir peticiones CRUD y obtener estadísticas de transacciones por cliente.

Consta de cinco servicios:

- **List Client:** Este servicio permite obtener la lista de clientes.
- **Get Client By Id:** Este servicio permite obtener un cliente a partir de su identificador.
- **Create Client:** Este servicio permite la creación de un cliente.
- **Update Client:** Este servicio permite actualizar un cliente .
- **Get Stats By Client Id:** Este servicio permite obtener las estadísticas diarias basado en las transacciones.

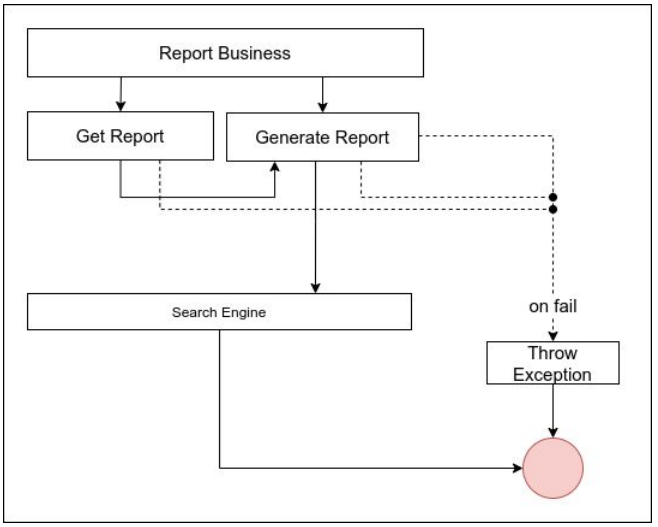


Fig. 7. Modelo de negocio del Reporte.

Este componente contiene la lógica de negocio para la generación de reportes, su función principal es generar reportes dinámicos de forma asíncrona.

- **Get Report:** Este servicio recibe las peticiones de generación de reportes, las envía al generador y devuelve la url del archivo a generar
- **Generate Report:** Este servicio genera asíncronamente los reportes y los guarda en S3 la ruta especificada por Get Report.

VIII. ARQUITECTURA FÍSICA

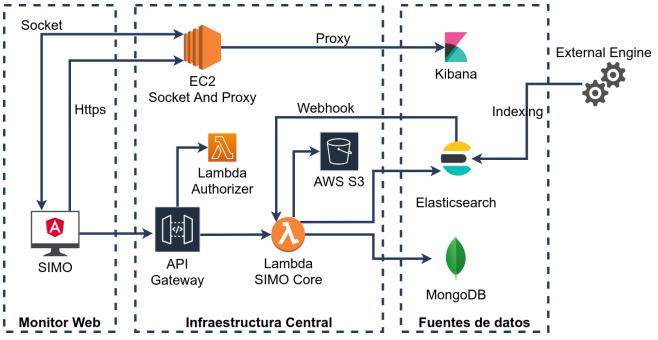


Fig. 8. Modelo de despliegue.

SIMO se divide en 3 grandes componentes (*ver fig .8)*:

1. Monitor Web: Se construyó sobre Angular debido a que es un framework que tiene compatibilidad con la mayoría de navegadores, posee un buen soporte y documentación robusta.

2. Infraestructura central: Se desplegó sobre *AWS* debido a su facilidad de uso, flexibilidad y alto desempeño. Los servicios usados fueron:

API Gateway: En este servicio se despliegan 3 APIs Rest:

1. **SimoSecurityManagementAPI:** Se integra con el servicio de seguridad que está desplegado en una función lambda.
2. **SimoCoreAPI:** Se integra con la arquitectura central.
3. **SimoMonitorAPI:** Se integra con las dos API anteriores, para ser el punto de entrada al sistema.

AWS Lambda: En este servicio se despliegan 3 funciones.

1. **SimoAuthorizer:** En esta función se implementa un esquema de autorización personalizado que usa la estrategia de autenticación por token.
2. **SimoSecurityManagement:** Este microservicio se encarga de administrar las entidades de seguridad: Usuarios, Roles, Grupos, Permisos y Tokens.
3. **SimoCore:** Este es el módulo central que contiene las lógicas de negocio de los componentes.

EC2: En este servicio se despliega Monstache y un Nginx Proxy.

1. **Monstache:** Es una librería de sincronización escrita en Go que indexa continuamente las colecciones de MongoDB de **Omnix** en Elasticsearch. Para el esquema del sincronizador se creó la siguiente jerarquía de directorios.

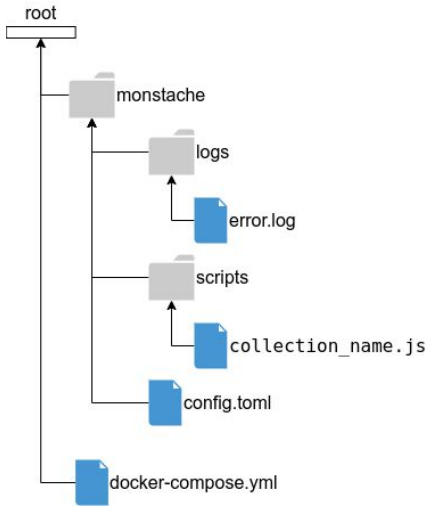


Fig. 9. Jerarquía de directorios.

Dentro de ***docker-compose.yml*** se encuentran los contenedores de cada dominio de datos. El archivo ***config.toml*** permite configurar el pipeline para la sincronización de los datos. La carpeta ***scripts*** contiene los middleware de cada colección, estos permiten transformar, eliminar, o agregar metadatos a los documentos a indexar. Finalmente, la carpeta de ***logs*** contiene los archivos para la depuración de errores de indexación.

2. **Nginx Proxy:** Permite el acceso desde el monitor hacía los dashboard en Kibana.

S3: En este servicio se guardan los reportes, snapshots de los índices de cada cliente, además funciona como hosting del *monitor web*.

1. **Reportes:** Se almacenan los reportes en S3 por su fácil administración, y simpleza en la transferencia de datos.
2. **Hosting:** Se usa el static Web Hosting por la fácil integración, alta disponibilidad y bajo coste.
3. **Snapshots:** La estrategia consiste en mantener los datos en diferentes clases de almacenamiento según su antigüedad en S3 como se indica en el siguiente gráfico (*ver fig. 10*):

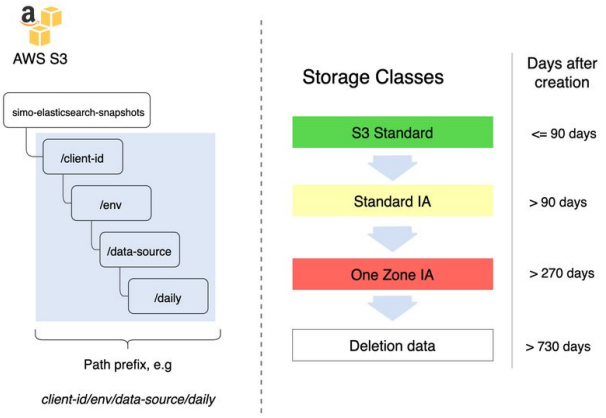


Fig. 10. Estrategia de ciclo de vida.

Para cada cliente se debe crear una carpeta en la raíz del bucket, y para cada una de estas carpetas se puede crear uno o más ciclos de vida dependiendo de los datos que esté almacenando.

- **S3 Standard:** En esta clase se mantendrán los archivos cuya antigüedad en S3 sea menor o igual a 90 días.
- **Standard IA:** Los datos mayores a 90 días pasarán a esta clase donde se mantendrán hasta que cumplan 270 días de antigüedad en S3.
- **One Zone IA:** Los datos mayores a 270 días pasarán a esta clase donde se mantendrán hasta que cumplan 730 días de antigüedad en S3.
- **Delete:** Los datos se eliminarán cuando cumplan más de 730 días de antigüedad en S3.

SNS: Este servicio permite publicar notificaciones de obtención de reportes para que el servicio de generación que está suscrito al topic específico, se encargue de procesar la información.

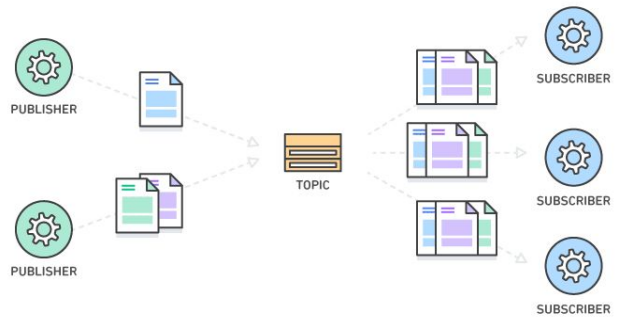


Fig. 11. SNS Topic.

3. Fuentes de datos: Permiten la gestión de la información del sistema.

1. **Elasticsearch:** Es un motor de búsqueda y análisis distribuido que permite una búsqueda y análisis en tiempo real para todos los tipos de datos, texto estructurado y no estructurados, datos numéricos, o geoespaciales. Elastic usa una estructura de datos llamado índice invertido el cual soporta búsquedas muy rápidas de texto de grandes volúmenes de datos. El estándar que se definió para los nombres de índices para los índices es el siguiente:

Fig. 12. Formato de nombres de índices.

Este formato permite saber cual es el cliente propietario de los índices, el entorno en el que está desplegado y fecha de creación.

2. **MongoDB:** Es una base de datos no relacional distribuida basada en documentos. La principal razón de esta elección es su alta disponibilidad, su eficiencia en el escalamiento horizontal y su versatilidad, ya que permite incluir cambios sobre la forma en que ingresan los datos sin necesidad de alterar su estructura esto supone flexibilidad en la gestión de las entidades transversales, las cuales son:
    a. **Clients:** Esta entidad contiene la información de cada cliente.
    b. **Notifications:** Esta entidad almacena las notificaciones fallidas de cada cliente.
    c. **Users:** Esta entidad contiene la información de cada usuario.
    d. **Role:** Esta entidad contiene los roles asignables a cada usuario.
3. **Kibana:** Es una interfaz de usuario que permite visualizar los datos de Elasticsearch.

XIII. Marco Teorico

Los microservicios son un patrón de arquitectura de software, utilizado para el desarrollo de proyectos en los que el sistema está compuesto por pequeños servicios independientes comunicados entre sí, a través de API's definidas. [22] Este patrón permite una mayor escalabilidad y mejor manejo de las aplicaciones al momento de desarrollar cada componente del proyecto. Los microservicios tienen un papel importante en el manejo y monitoreo de los sistemas transaccionales, dado que estos representan una alternativa a la arquitectura monolítica, ofreciendo un bajo acoplamiento (Independencia entre los servicios) y una alta cohesión (Tareas específicas para cada servicio)[23]. Por consiguiente, los microservicios permiten crear aplicaciones en donde cada proceso de las mismas es un servicio independiente, capaz de comunicarse con los demás componentes, pero sin interferir en estos.

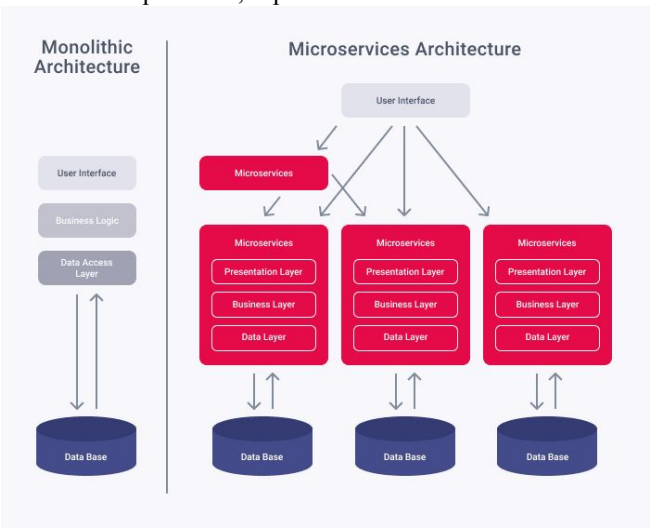


Fig. 13. Arquitectura monolítica vs microservicios.

En [14] señalan que los fallos en las aplicaciones son inevitables, incluso en aquellos sistemas diseñados especialmente para resistir este tipo de inconvenientes en la disponibilidad del servicio; todo esto a través del uso de mecanismos de tolerancia ante fallos. Por tal razón, al diseñar una aplicación se hace indispensable tener presente la tolerancia de fallos y el mecanismo de tolerancia de estos como factor primordial para garantizar una alta disponibilidad.

A summary of recent failure events experience by popular cloud services.

Cloud Services	Down-time	Impact
Tencent Cloud (2018.07.24)[1]	2 hours	Users were unable log on to Tencent Cloud platform; large amounts of data of startups were lost.
Ali Cloud (2018.06.27)[2]	0.5 hours	Taobao and Tmall backend were interrupted; transaction records of some of the orders went missing.
Amazon AWS (2017.02.28)[3]	5 hours	a large set of servers was inadvertently removed, affecting nine services; countless cloud-based applications and websites offline.
Google Cloud (2016.08.11)[4]	1.5 hours	improper operation of balance traffic led to the outage of Google App Engine; error rates and latency of services were elevated.
Apple iCloud (2015.05.21)[5]	11 hours	iCloud services were affected, including Cloud Mail, iMovie, Photos, Documents, et al.; 500 million iCloud members and the estimate to 200 million users impacted.

Fig. 14. Arquitectura monolítica vs microservicios.

Además, en la tabla anteriormente mostrada, se pueden notar algunos fallos que se presentaron en servicios populares basados en la nube, mostrando así el impacto que tienen estos fallos en los servicios y las grandes pérdidas que pueden representar para una compañía. De allí se puede afirmar que es de igual importancia tanto el correcto diseño del mecanismo de tolerancia de fallos, así como verificar que el mecanismo de tolerancia cumpla con su propósito de la manera esperada.

En segundo lugar, se identifica que el "Modelo de Dominio" (Domain Driven Design) está fuertemente relacionado a los microservicios, el cual es un enfoque que representa las relaciones entre las entidades, así como sus atributos, brindándonos una visión íntegra del alcance de un proyecto [24]. Esta técnica se encuentra estructurada por distintas prácticas que permiten que el equipo de trabajo tome decisiones de diseño que posibiliten enfocar y acelerar el manejo de los dominios complejos de nuestro sistema. Por ende, el equipo de trabajo logra tener conocimiento sobre los límites de su aplicación, evitando la creación de aplicaciones funcionales, carentes de sentido a la hora de entenderlas o hacerlas funcionar. En [15] afirman que el acercamiento propuesto en este modelo facilita a los desarrolladores reducir la complejidad de un servicio al involucrar a expertos en el tema en el proceso de desarrollo. El objetivo principal del modelo es la agrupación sistemática de todos los requerimientos en un modelo de dominios realista, implementable y funcional, con el objeto de refinar y adaptar el conocimiento adquirido sobre el proyecto, ofreciendo soluciones técnicas que eviten el sobrecargo de los desarrolladores. De igual forma, este modelo se encuentra definido como un lenguaje común que hace posible la comunicación entre los desarrolladores y los expertos del dominio, evitando las malas interpretaciones precisadas anteriormente. En pocas palabras, este modelo facilita sustancialmente la tarea obtener un número adecuado de micro servicios funcionales.

Por otra parte, tenemos los sistemas basados en transacciones (**OMS**); estos son sistemas de negocios que permiten recopilar información de las transacciones y guardarla en una base de datos, permitiendo así que los usuarios hagan uso de esta información. Este tipo de sistema es muy utilizado en los casos en los que se necesita vender o comprar cualquier objeto, ya sea una orden de venta, una reserva en un hotel o la compra de un supermercado[25]. Sin embargo, este tipo de sistemas también puede utilizarse en otros conceptos, como el manejo de la información de las personas que hacen parte de una compañía, los datos de fabricación de cierto lote de productos o los datos de transporte de mercancías. En resumen, este tipo de sistema es altamente utilizado para realizar operaciones comerciales.

Los sistemas de alerta temprana(SAT/EWS por sus siglas en inglés) son mecanismos independientes del sistema principal, cuya función es alertar cambios o fallos en el mismo. Dichos sistemas son usados mayormente para monitorear fenómenos ambientales, evitando catástrofes y desastres mayores, reduciendo así pérdidas económicas, materiales y humanas[26]. En [16] sostienen que las condiciones climáticas extremas a menudo causan daños considerables, los cuales podrían ser mitigados si las personas estuvieran mejor preparadas para los desastres futuros. Por eso, las autoridades públicas y privadas han estado invirtiendo dinero en los sistemas de alerta temprana durante los últimos años"[27]. Con ese orden de ideas, este tipo de sistemas es tomado como modelo para el desarrollo del sistema de monitoreo, pues se plantea implementar un sistema de alertas basado en subdivisiones de microservicios, tal como se ha expuesto anteriormente, de modo que se asimile así el funcionamiento de un SAT. Como resultado, se le otorga al usuario acceso temprano y en tiempo real a los errores y las anomalías del sistema, para así poder actuar de manera oportuna ante estos, evitando que escalen y se vuelvan una problemática mayor.

Otro punto a considerar es la computación en la nube (Cloud Computing), que se puede definir como proveer o suministrar recursos informáticos a los clientes por medio de un modelo de pago periódico, brindándoles a estos software, plataformas o infraestructura para sus proyectos, es decir, se refiere a la gestión de los recursos de TI que se tienen en un espacio de desarrollo virtual[28]. La computación en la nube nos lleva a una implementación más rápida y eficiente de mejoras y otros avances tecnológicos [17]. Por ello es notorio que este método beneficia en gran medida los sistemas transaccionales, pues brinda una mayor escalabilidad, disponibilidad y seguridad para el sistema en general, los cuales son factores muy importantes si se tiene en cuenta la sensibilidad de la información que se maneja con los OMS, así como también la cantidad de datos que se manejan en estos. Además de esto, se tienen los servicios web como un componente importante del servicio almacenado en la nube (Cloud Computing), pues permiten la comunicación entre los distintos servicios proporcionados. Estos se basan en el envío y recepción de solicitudes y respuestas entre un cliente y un servidor, entendiéndolo así como un intercambio de 'mensajes' entre un emisor y un receptor[29]. Cabe resaltar que los servicios web representan un pilar fundamental para los sistemas transaccionales, pues estos permiten acceder a toda la información que se genera durante la ejecución de los servicios.

Por último, tenemos la seguridad como un pilar fundamental en el desarrollo de las compañías, pues actualmente los fallos del sistema pueden dañar la reputación de las compañías, poniendo en juego la fidelidad de los clientes y el valor de mercado que estos representan [18]. Por tanto, podemos entender la seguridad informática

como toda aquella técnica que garantiza la permanencia e integridad de la información que se almacena en un computador. Además de esto, existen muchos protocolos para el envío seguro de la información a través de internet. Por ejemplo, tenemos los firewalls o la encriptación. Luego, en cuanto al monitor de transacciones tenemos la autorización, revisión de las cuentas, confidencialidad, integridad, disponibilidad y el no repudio[30]. Todos estos factores deben ser tenidos en cuenta al momento de implementar el sistema de seguridad del sistema.

IV. Síntesis de la revisión sistemática de la literatura

Artículo	**Año**	**Palabras claves**	**Fuente**
Mobile Technology Contributing to Omni-Channel Retail	2018	E-commerce, Omnichannel, Retail	ACM
Microservices: A Performance Tester's Dream or Nightmare	2020	Microservices,Performance	ACM
Performance Engineering for Microservices: Research Challenges and Directions	2017	Microservices, performance, scalability, availability, portability	ACM
Learning to log helping developers make informed logging decisions	2015	Logging, Software systems	ACM
Benchmarking Microservice Performance: A Pattern-based Approach	2020	Restful web services, performance	ACM
Where Do Developers Log? An Empirical Study on Logging Practices in Industry	2014	Automatic logging	ACM
Engineering Configurators for the Retail Industry	2018	Configuration, Retail	ACM
Managing eCommerce Service Failures: Incorporating Validity, Provenance and Trust from the Semantic Web		Semantic web, information validity, Web service failure	ACM
Infrastructure Cost Comparison of Running Web Applications in the Cloud using AWS Lambda and Monolithic and Microservice Architectures	2016	Computer architecture, Service-oriented architecture, Cloud computing, Companies, Time factors, Logic gates	ACM
Test Coverage Criteria for RESTful Web APIs	2016	REST, Testing web, services	ACM
Resource-based Test Case Generation for RESTful Web Services	2019	Search-based Test Case Generation, RESTful Web Service Testing	ACM
Building Bridges - The System Administration Tools and Techniques Used to Deploy Bridges	2017	High performance computing, Node deployment	ACM
Tunable Consistency in MongoDB	2019	MongoDB	ACM
An Extensible Fault Tolerance Testing Framework for Microservice-based Cloud Applications	2018	Microservices,Fault tolerance testing, Fault injection	ACM
Partitioning Microservices: A Domain Engineering Approach	2018	Sizing microservice, DDD pattern, weather domain	ACM

An Early Warning System for Suspicious Accounts	2017	Online services, Machine learning	ACM
Cloud Computing Vulnerabilities Analysis	2017	cloud computing; vulnerabilities; CVE; CVSS; security	ACM
Information Securing in Organizations: A Dialectic Perspective	2019	Information Security, Organizational Structure, Dialectics	ACM
Mobile Technology Contributing to Omni-Channel Retail	2018	Omni-Channel, Channel Integration, Mobile Commerce, Mobile Channel, Mobile Technology, Omni-Channel Retailing, Online Channel, Retailing	ACM
AutoMAP: Diagnose Your Microservice-based Web Applications Automatically	2020	Microservice architecture, web application, anomaly diagnosis, root cause, cloud computing	ACM
Connectedness Testing of RESTful Web-Services	2010	Representational State Transfer, Specification based Testing, Testing, Web-service	ACM
Microservices for Scalability	2016	Microservices, scalability, monitoring	ACM
Contextual understanding of microservice architecture: current and future directions	2018	microservice, architecture	ACM
Microservice architecture and model-driven development: yet singles, soon married (?)	2018	Model-driven software engineering, Cloud computing, Domain specific languages	ACM
The Design and Implementation of a Process-Based Printing Order Management System	2013	Printing, Production, Computational modeling, Industries, Software, Companies	IEEE
Risk Early Warning Index System in the Field of Public Safety in Big Data Era	2015	Indexes, Monitoring, Hazards, Security	IEEE
Digital governance and hotspot geoinformatics for monitoring, etiology, early warning, and management around the world	2006	microservice architecture, model-driven development, model-driven microservice development, model transformation, modeling languages, domain-driven design	ACM
Cloud computing: state of the art and security issues	2015	Cloud Computing, Cloud Architecture, Entities.	ACM
Case Study on Data Communication in Microservice Architecture	2019	Microservices, Cloud-computing, System Integration	ACM

Security Maturity Model of Web Applications for Cyber Attacks	2019	Penetration tests; Vulnerability of the web application; Security information; Web attack; Cyber-attack; Cyber defense; Clinical records	ACM

IX. PROTOTIPO

Para este proyecto, se busca realizar la implementación de un prototipo funcional que cumpla las especificaciones y objetivos establecidos previamente por el equipo de trabajo. Para esto se desplegó el Monitor Web, sobre Angular el cual se presenta a continuación, con las especificaciones de cada componente visual.

Fig. 15. Landing Page.

El propósito del landing page (*ver fig. 15*) es llamar la atención del usuario para así conseguir clientes potenciales o ayudar a fidelizar a los clientes antiguos, presentandoles un estilo minimalista pero sin perder de vista la información relevante del sistema.

Componentes visuales:

1. Logo de SIMO.
2. Enlace con la landing page que redirige hacia la parte informativa del proyecto.
3. Botón de login del usuario en el sistema.
4. Presentación inicial con el eslogan y mensaje del monitor.
5. Información relevante del sistema de monitoreo.

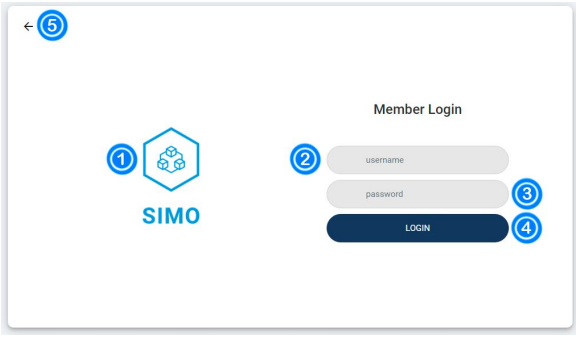


Fig. 16. Login del sistema.

El login del sistema (*ver fig. 16*) es utilizado para solicitar al usuario sus credenciales, con el fin de autenticar y validar su identidad.

Componentes visuales:

1. Logo del proyecto.
2. Nombre de usuario requerido para el inicio sesión.
3. Contraseña requerida para la validación del usuario.
4. Botón de inicio de sesión que redirige al usuario a la vista de clientes.
5. Botón de retroceso que permite al usuario volver al landing page.

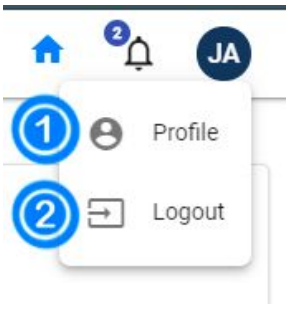


Fig. 17. Menú del usuario.

Este menú desplegable (*ver fig. 17*) se utiliza para recopilar las acciones que conciernen al usuario, como el acceso a su perfil o el logout del sistema.

Componentes visuales:

1. Enlace a la información del perfil del usuario.
2. Botón para cerrar sesión y volver a la vista del login de usuarios.

Fig. 18. Perfil del usuario.

Esta vista (*ver fig. 18*) Se utiliza para resumir la información relevante del usuario logueado en el sistema.

Componentes visuales:

1. Nombre del usuario logueado.
2. Información de la responsabilidad del usuario.
3. Nombre del cliente en que se encuentra ubicado el usuario.
4. Rol del usuario.

Fig. 19. Barra de estado.

Está barra de estado (*ver fig. 19*) se utiliza para darle al usuario un acceso rápido a las distintas pestañas del sistema.

Componentes visuales:

1. Ícono de inicio para redirigir al usuario hacia la vista de gestión de clientes.
2. Menú desplegable en el que se muestran las notificaciones que recibe el sistema.
3. Menú desplegable que contiene información del usuario y de logout.

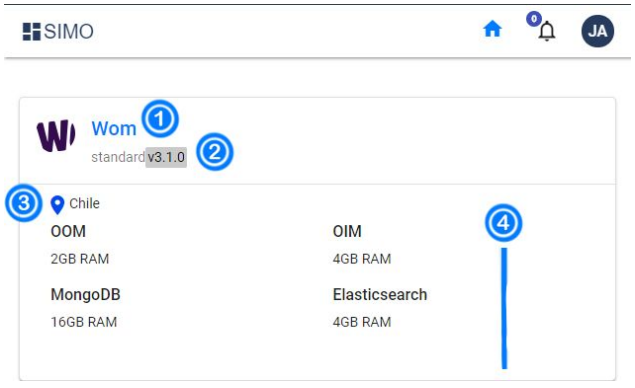


Fig. 20. Gestión de clientes.

En esta vista (*ver fig. 20*) se busca resumir la información de recursos de un cliente así como información básica del mismo, todos los clientes serán desplegados en esta pestaña.

Componentes visuales:

1. Ícono y nombre del cliente.
2. Versión desplegada de Omnix.
3. País en el que se encuentra ubicado el cliente.
4. Recursos asignados al componente de despliegue.

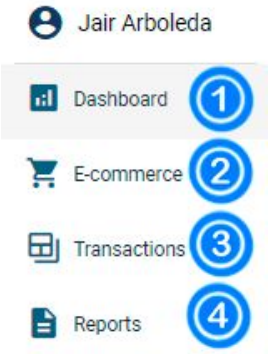


Fig. 21. Menú del sistema.

Este menú principal (*ver fig. 21*) contiene las distintas funcionalidades del sistema.

1. Enlace a la vista del dashboard transacciones del cliente.
2. Enlace a la vista del dashboard del ‘E-commerce’.
3. Enlace a los registros de transacciones del cliente.
4. Enlace a los reportes generados por el usuario.

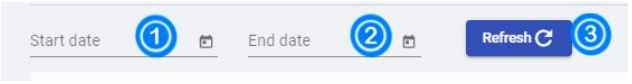


Fig. 22. Filtros de búsqueda.

Controles utilizados para filtrar por rangos de fecha. Este componente (*ver fig. 22*) se encuentra presente en las vistas listadas en la *fig.* 23-24-25-26

1. Cota inferior para el filtrado de transacciones por fecha.
2. Cota superior para el filtrado de transacciones por fecha.
3. Botón para actualizar la información del con los filtros establecidos.

Fig. 23. Dashboard de transacciones.

Esta es la vista principal del dashboard del cliente (*ver fig. 23*), donde se muestra de manera visual las estadísticas

del mismo. Además, permite filtrar esta información con distintos parámetros.

Componentes visuales:

1. Visualización de las estadísticas desplegadas por el dashboard del cliente.

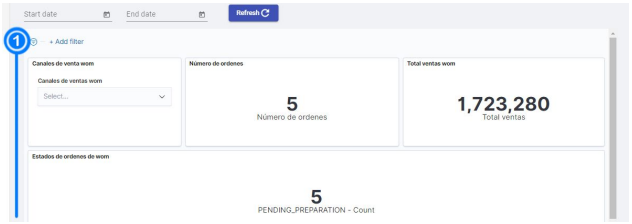


Fig. 24. Dashboard de E-commerce.

Está vista (*ver fig. 24*) se creó para mostrar la información comercial del cliente de manera escrita y gráfica para brindar facilidad de lectura.

Componentes visuales:

1. Dashboard en el que se presentan las estadísticas comerciales del cliente.

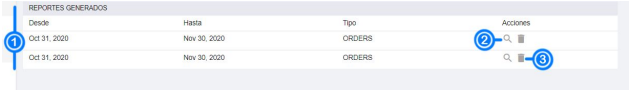


Fig. 25. Vista de reportes

Esta vista (*ver fig. 25*) se utiliza para el despliegue de los reportes generados por el sistema, permitiendo al usuario descargarlos o eliminarlos.

Componentes visuales:

1. Presentación de los reportes generados por el sistema.
2. Botón para descargar el reporte.
3. Opción de eliminación del reporte.

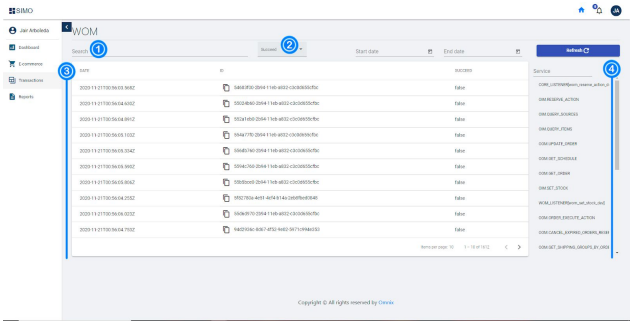


Fig. 26. Vista de transacciones

Está vista (*ver fig. 26*) representa un componente de depuración de transacciones que permite realizar búsquedas basándose en el identificador de una orden, el estado de la misma o permite utilizar un intervalo de fechas como parámetro para las búsquedas.

Componentes visuales:

1. Barra de búsqueda de transacciones.
2. Menú de selección del estado de la transacción.
3. Lista de servicios elegibles del sistema.
4. Listado de transacciones consultadas al aplicar los filtros de búsqueda, estas pueden ser seleccionadas para visualizar información detallada(*fig. 27-28*)

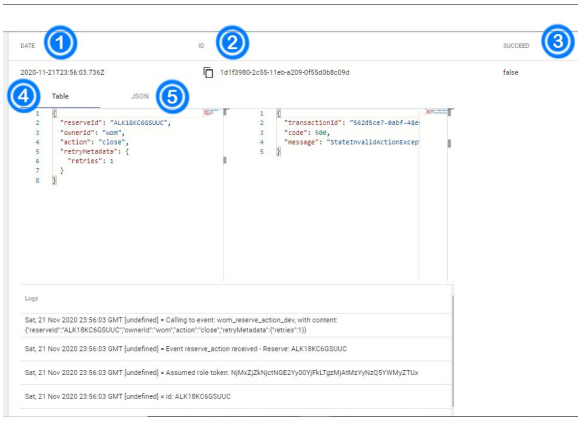


Fig. 27. Vista formato tabla de la transacción.

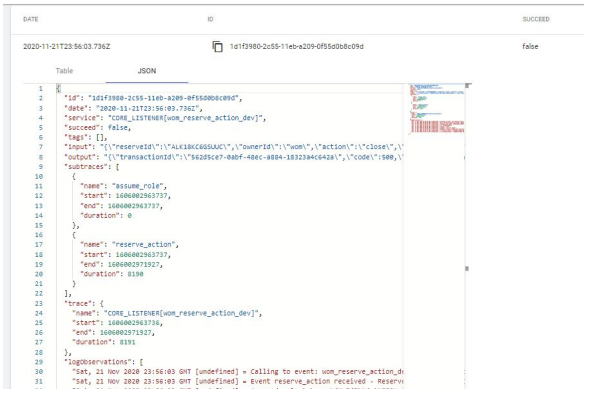


Fig. 28. Vista formato JSON de la transacción.

Estas vistas (*ver fig. 27-28*) se utilizan para desplegar información detallada de las transacciones luego de ser consultadas con cada uno de los filtros.

Componentes visuales:

1. Fecha de creación de la transacción consultada.
2. ID asociado a la transacción.
3. Estado de la transacción, (exitosa o fallida)
4. Botón para seleccionar el formato de tabla para la visualización de la transacción.
5. Botón para seleccionar el formato JSON para la visualización de la transacción.

X. VALIDACIÓN

En el ámbito de la asignatura Proyecto Final dos grupos de pares realizaron la validación del prototipo usando el estándar ISO 15504. De esta manera se determinó la calidad del producto desarrollado y el resultado de dicha evaluación se puede observar en la *fig 29*.

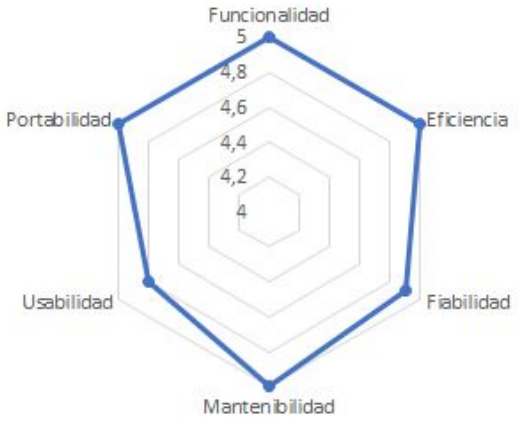


Fig. 29. Validación del prototipo

XI. RESULTADO

El Sistema de Monitoreo de Omnix cumplió con los objetivos propuestos y de manera general el diseño de la micro-arquitectura suple todos los requerimientos establecidos. Después de la fase de despliegue se realizó un histórico de las principales peticiones al área de soporte con el propósito de poseer métricas del funcionamiento y respuesta del sistema. Por este motivo se decidió tomar una muestra de tickets comprendidos entre Agosto 22 y Noviembre 14 con la finalidad de demostrar el impacto del sistema de monitoreo en la resolución de incidencias. El 25 de septiembre se desplegó el sistema y por ello se fraccionaron las gráficas en dos secciones para diferenciar el comportamiento de los datos antes y después del despliegue de SIMO con la finalidad de verificar el cumplimiento de los objetivos. Se decidió realizar 4 gráficas comparativas las cuales siguen a continuación:

Support requests summary

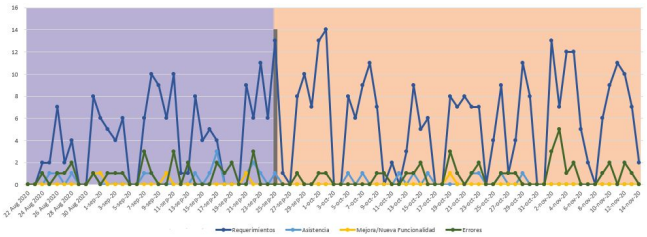


Fig. 30. Gráfica de solicitudes de soporte (Requerimientos, Asistencia, Mejoras, Errores) (: Antes del despliegue, : Después del despliegue)

En la (*fig. 30*) se evidencia que en ambas etapas graficadas la creación de tickets posee un comportamiento regular y constante. Es decir, que el número de tickets se mantuvo constante antes y después del despliegue.

Created vs Resolved summary

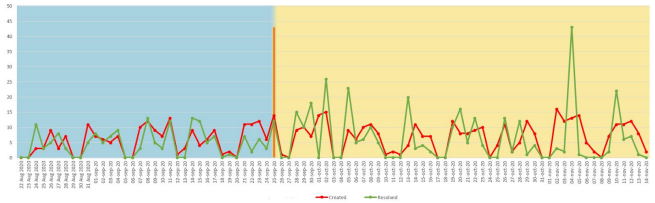


Fig. 31. Gráfica ticket creados vs resueltos(: Antes del despliegue, : Después del despliegue)

En la (*fig. 31*) a pesar de que la creación de tickets se mantiene constante, se evidencia que posterior al despliegue los tickets resueltos presentan un leve aumento, corroborando así que el sistema optimiza el proceso de detección de fallos generando respuestas más rápidas por parte del equipo de soporte.

SLA Success rate summary

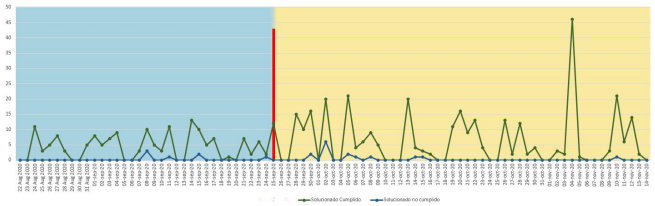


Fig. 32. Gráfica de éxito de SLA(: Antes del despliegue, : Después del despliegue)

En la (*fig. 32*) se observa, que la proporción de cumplimiento de los SLAs en los tickets solucionados posteriores al despliegue, tuvieron un leve aumento en comparación a las proporciones antes del despliegue. Dando a entender que la solución fue beneficiosa para los SLAs.

Bug report summary

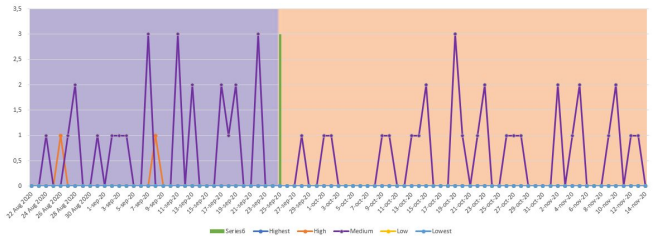


Fig. 33. Gráfica de reporte de bugs clasificados por su gravedad (High, Medium, Low) (: Antes del despliegue,, : Después del despliegue).

En la (*fig. 33*) se puede visualizar que la etapa anterior al despliegue el número de reportes de bugs por parte de los clientes está bastante acoplado en comparación a la etapa posterior la cual presenta una menor frecuencia de reportes de bugs, una posible causa de dicho patrón se podría relacionar con el manejo inadecuado de los servicios de Omnix por parte de los clientes los cuales ocasionalmente generan reportes de bug sin existir un fallo real en el sistema (*falsos positivos)*.

XII. Conclusiones

El monitoreo es una parte fundamental de la administración de los sistemas debido a que brinda un marco para la toma de decisiones tempranas para así lograr anticiparse y optimizar el uso de los recursos disponibles. El proyecto no solamente funciona como monitor de soporte de Omnix, también posee un gran potencial de cara a los clientes que no cuentan con un sistema de monitoreo de este tipo, por lo que el concepto se puede visualizar como un producto multifuncional. Esto abre un abanico de posibilidades en cuanto a lo que como producto puede llegar a ser el Sistema de Monitoreo de Omnix.

La solución es un producto potencial en el marco del monitoreo a alto y bajo nivel de e-commerce y/o sistemas gestores de órdenes, que no tienen referentes o antecedentes locales, como lo comenta Edwin Vargas, CTO de Omnix, el OMS que inspiró este proyecto y la puesta en marcha de la propuesta con la finalidad de centralizar las transacciones para optimizar los tiempos de consulta, mejorar la monitorización y el abordaje de las incidencias en Omnix. Para cumplir este objetivo, se siguió un sistema metodológico de 6 fases, con el fin de desarrollar y obtener los objetivos planteados. Esto dio paso al desarrollo del proyecto, trayendo consigo retos de implementación y despliegue como, por ejemplo: la integración de los microservicios, la correcta implementación del modelado de 3 capas, y finalmente, la correcta implementación de las vistas del monitor. Por otro lado, los resultados obtenidos fueron prometedores, dado que se logró mejorar los tiempos de respuestas, aumentar la satisfacción de los clientes, , mejorar la proporción de tickets resueltos y reducir los tiempos de solución de los SLAs.

Finalmente, esta propuesta se plantea como una solución real y a la vanguardia de los sistemas basados en transacciones dando lugar al Sistema de Monitoreo de Omnix o SIMO que se presenta como un producto potencial en el marco del monitoreo de e-commerce y/o sistemas de gestión de órdenes.

Referencias

[1] Andreas Mladenow, Antoaneta Mollova, and Christine Strauss. 2018. Mobile Technology Contributing to Omni-Channel Retail. In Proceedings of the 16th International Conference on Advances in Mobile Computing and Multimedia (2018). Association for Computing Machinery, New York, NY, USA, 92–101. DOI:https://doi.org/10.1145/3282353.3282371

[2] Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dušan Okanović, and André van Hoorn. 2020. Microservices: A Performance Tester's Dream or Nightmare In Proceedings of the ACM/SPEC International Conference on Performance Engineering(ICPE '20). Association for Computing Machinery, New York, NY, USA, 138–149. DOI:https://doi.org/10.1145/3358960.3379124

[3] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. 2017. Performance Engineering for Microservices: Research Challenges and Directions. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (ICPE '17 Companion). Association for Computing Machinery, New York, NY, USA, 223–226. DOI:https://doi.org/10.1145/3053600.3053653

[4] J. Zhu, P. He, Q. Fu, H. Zhang, M. Lyu, D. Zhang. (May 2015). Learning to Log: Helping Developers Make Informed Logging Decisions. [Online]. Available: : https://ezproxy.uninorte.edu.co:2424/doi/10.5555/2818754.2818807

[5] M. Grambow, L. Meusel, D. Bermbach.(March, 2020). Benchmarking Microservice Performance: A Pattern-based Approach. [Online] Available: https://ezproxy.uninorte.edu.co:2424/doi/10.1145/3341105.3373875

[6] Q. Fu, J. Zhu, W. Hu, J. Lou, R. Ding, Q. Lin, D. Zhang, T. Xie. (May 2014). Where Do Developers Log An Empirical Study on Logging Practices in Industry. [Online]. Available: https://ezproxy.uninorte.edu.co:2424/doi/10.1145/2591062.2591175

[7] Maxime Cordy and Patrick Heymans. 2018. Engineering configurators for the retail industry: experience report and challenges ahead. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18). Association for Computing Machinery, New York, NY, USA, 2050–2057. DOI:https://doi.org/10.1145/3167132.3167352

[8] M. Fox. [August 2012]. Managing ecommerce service failures: incorporating validity, provenance and trust from the semantic web. [Online] Available: https://dl.acm.org/doi/10.1145/2346536.2346553

[9] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, M. Lang. (May 2016). Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. Available: https://dl.acm.org/doi/10.1109/CCGrid.2016.37

[10] A. Lopez, S. Segura, A. Cortés. (August 2019). Test coverage criteria for RESTful web APIs. [Online]. Available: https://ezproxy.uninorte.edu.co:2424/doi/10.1145/3340433.3342822

[11] M. Zhang, B. Marculescu, A. Arcuri. Resource-based test case generation for RESTful web services. [Online]. Available: https://dl.acm.org/doi/10.1145/3321707.3321815

[12] R. Underwood. (July 2019). Building Bridges - The System Administration Tools and Techniques Used to Deploy Bridges. [Online]. Available: https://ezproxy.uninorte.edu.co:2424/doi/pdf/10.1145/3093338.3093339

[13] W. Schultz, T. Avitabile, A. Cabral (2019). Tunable Consistency in MongoDB. [Online]. Available: https://dl.acm.org/doi/10.14778/3352063.3352125

[14] N. Wu, D. Zuo, Z. Zhang. (November 2018). An extensible fault tolerance testing framework for microservice-based cloud applications. [Online]. Available: https://ezproxy.uninorte.edu.co:2424/doi/10.1145/3290420.3290476

[15] M. Immaculée, J. Doreen, T. Mukasa, B. Kanagwa, J. Balikuddembe. (May 2019). Partitioning Microservices: A Domain Engineering Approach. [Online]. Available: https://ezproxy.uninorte.edu.co:2424/doi/10.1145/3195528.3195535

[16] H. Halawa, M. Ripeanu, K. Beznosov, B. Coskun, M. Liu .(November 2017) .Early warning systems in practice: performance of the SAFE system in the field. [Online]. Available: https://dl.acm.org/doi/10.1145/3128572.3140455

[17] A. Zamfiroiu, I. Petre, R. Boncea. (September 2019). Cloud Computing Vulnerabilities Analysis. [Online]. Available: https://ezproxy.uninorte.edu.co:2424/doi/10.1145/3361821.3361830

[18] Y. Li, T. Stafford, B. Fuller, S. Ellis. (). Information Securing in Organizations: A Dialectic Perspective. [Online].

[19] Andreas Mladenow, Antoaneta Mollova, and Christine Strauss. 2018. Mobile Technology Contributing to Omni-Channel Retail. In Proceedings of the 16th International Conference on Advances in Mobile Computing and Multimedia (2018). Association for Computing Machinery, New York, NY, USA, 92–101. DOI:https://doi.org/10.1145/3282353.3282371

[20] M. Ma, P Wang, J. Xu, Y. Wang, P. Cheng, Z. Zhang (April 2020). AutoMAP: Diagnose Your Microservice-based Web Applications Automatically. [online]. Available: https://ezproxy.uninorte.edu.co:2424/doi/pdf/10.1145/3366423.3380111

[21] S. Kumar, R. Rodriguez. (February 2017) Connectedness Testing of RESTful Web-Services. [online]. Available: https://ezproxy.uninorte.edu.co:2424/doi/pdf/10.1145/1730874.1730902

[22] Wilhelm Hasselbring. (2016). Microservices for Scalability. Available: https://dl.acm.org/doi/pdf/10.1145/2851553.2858659?casa_token=de2KZ7iKDbMAAAAA:XKeKTBuadNkvftplaooXKZ27g2G5hYjKF2RAYK4o7U-jJm8EJQNU9kdFzMx8YGcf3JsF8goPIYEB4uk

[23] M. Trnka, M. Donahoo, T. Cerny. (Jan 2018). Contextual understanding of microservice architecture: current and future directions. [Online] .Available: https://dl.acm.org/doi/pdf/10.1145/3183628.3183631?casa_token=1DOBTrZhdq4AAAAA:GDpa04uQdzWqzS18DFMWQI37UCiZ6_aUW2MVFEvJ4uXfa0xi-aWU_487kPfjYaElbI4mq_zP_wPuGwk

[24] F. Rademacher, J. Sorgalla, P. Nils, S. Sachweh, A. Zundorf. (May 2018). Microservice architecture and model-driven development: yet singles, soon married. https://dl.acm.org/doi/pdf/10.1145/3234152.3234193?casa_token=g_KV2PxsIJoAAAAA:u6t335UXE4u2DHSA99bpdsZPI_XSdiITMF2hOKvC0ZAFLVzPF9W_QdrG1NnDX1TY2CtUUY10f4cZocw

[25] Huiling Ma; Yongbin Zhang. (2013). The Design and Implementation of a Process-Based Printing Order Management System. Available: https://ieeexplore.ieee.org/document/6973568

[26] Dongmei Cui. (2015). Risk Early Warning Index System in the Field of Public Safety in Big Data Era. Available: https://dl.acm.org/doi/10.1109/ISDEA.2015.180

[27] G.P PATIL .(May,2006). Digital governance and hotspot geoinformatics for monitoring, etiology, early warning, and management around the world. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/1146598.1146626

[28] Shruti Chhabra,V. S. Dixit. (2015). Cloud computing: state of the art and security issues. Available: https://ezproxy.uninorte.edu.co:2424/doi/pdf/10.1145/2735399.2735405

[29] A. Smid, R. Wang, T. Cerny. (Sept 2019). Case Study on DataCommunication in Microservice Architecture. [Online]. Available: https://ezproxy.uninorte.edu.co:2424/doi/pdf/10.1145/3338840.3355659

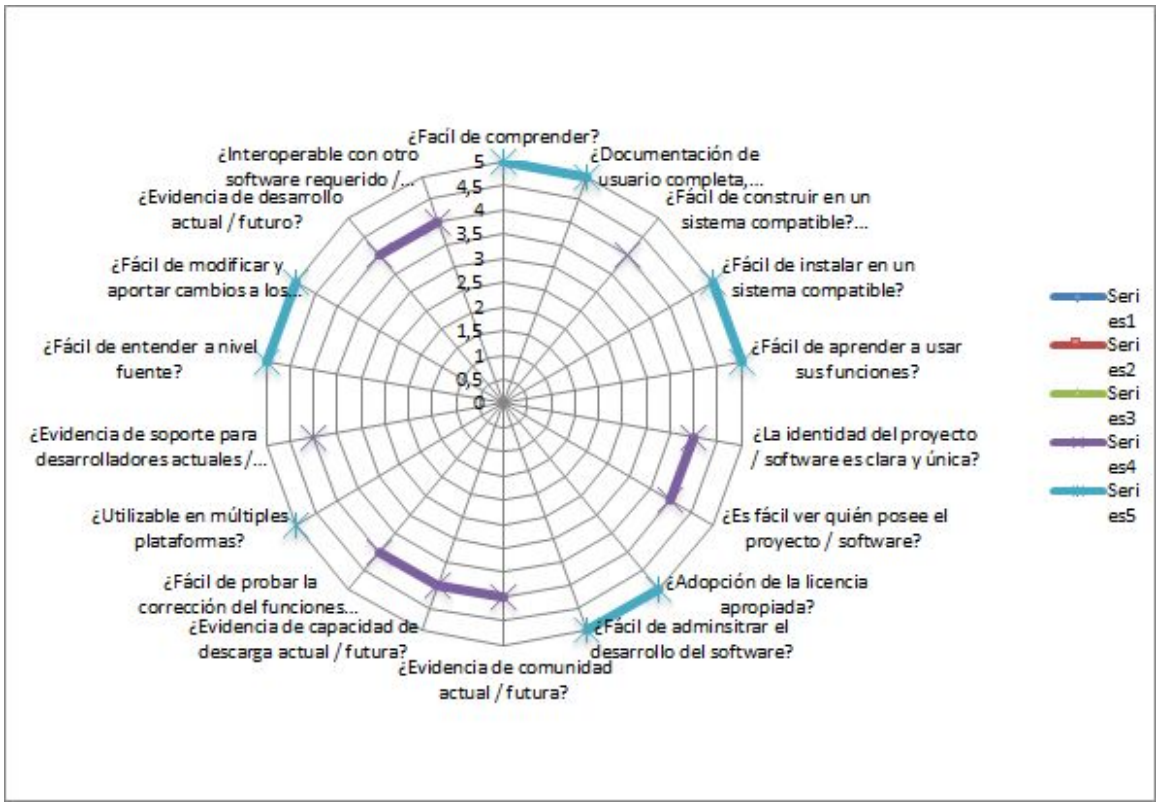
[30] Renato Rojas, Ana muedas, David Mauricio. (2019). Security maturity model of web applications for cyber attacks Available: https://dl.acm.org/doi/pdf/10.1145/3309074.3309096

Anexos

Calificación 1

Juan David Solano, Roger Andres Herazo; Roberto Acuña

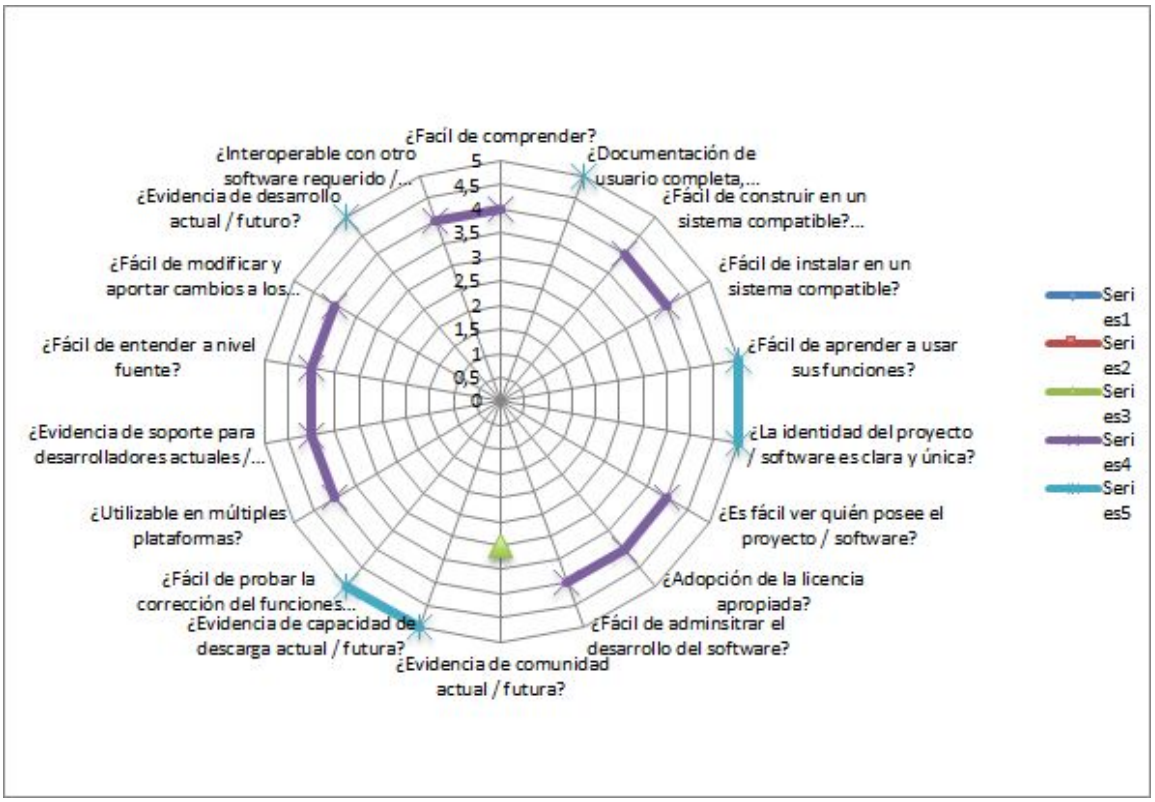
Caracteristica	Definición o descirpción	1	2	3	4	5
Understandability	¿Facíl de comprender?					5
Documentation	¿Documentación de usuario completa, apropiada y bien estructurada?					5
Buildability	¿Fácil de construir en un sistema compatible? (Close-Open)				4	
Installability	¿Fácil de instalar en un sistema compatible?					5
Learnability	¿Fácil de aprender a usar sus funciones?					5
Identity	¿La identidad del proyecto / software es clara y única?				4	
Copyright	¿Es fácil ver quién posee el proyecto / software?				4	
Licencing	¿Adopción de la licencia apropiada?					5
Governance	¿Fácil de adminsitrar el desarrollo del software?					5
Community	¿Evidencia de comunidad actual / futura?				4	
Accessibility	¿Evidencia de capacidad de descarga actual / futura?				4	
Testability	¿Fácil de probar la corrección del funciones caja negra?				4	
Portability	¿Utilizable en múltiples plataformas?					5
Supportability	¿Evidencia de soporte para desarrolladores actuales / futuros?				4	
Analysability	¿Fácil de entender a nivel fuente?					5
Changeability	¿Fácil de modificar y aportar cambios a los desarrolladores?					5
Evolvability	¿Evidencia de desarrollo actual / futuro?				4	
Interoperability	¿Interoperable con otro software requerido / relaciona				4	
modelo de evaluación basado en el estandar ISO 9126 ISO 15504 + ESCALA DE LIKERT						



Calificacion 2

Jesus Mendinueta, Hector Camacho, Javier Valencia

Caracteristica	Definición o descirpción	1	2	3	4	5
Understandability	¿Facíl de comprender?				4	
Documentation	¿Documentación de usuario completa, apropiada y bien estructurada?					5
Buildability	¿Fácil de construir en un sistema compatible? (Close-Open)				4	
Installability	¿Fácil de instalar en un sistema compatible?				4	
Learnability	¿Fácil de aprender a usar sus funciones?					5
Identity	¿La identidad del proyecto / software es clara y única?					5
Copyright	¿Es fácil ver quién posee el proyecto / software?				4	
Licencing	¿Adopción de la licencia apropiada?				4	
Governance	¿Fácil de adminsitrar el desarrollo del software?				4	
Community	¿Evidencia de comunidad actual / futura?			3		
Accessibility	¿Evidencia de capacidad de descarga actual / futura?					5
Testability	¿Fácil de probar la corrección del funciones caja negra?					5
Portability	¿Utilizable en múltiples plataformas?				4	
Supportability	¿Evidencia de soporte para desarrolladores actuales / futuros?				4	
Analysability	¿Fácil de entender a nivel fuente?				4	
Changeability	¿Fácil de modificar y aportar cambios a los desarrolladores?				4	
Evolvability	¿Evidencia de desarrollo actual / futuro?					5
Interoperability	¿Interoperable con otro software requerido / relaciona				4	
	modelo de evaluación basado en el estandar ISO 9126 ISO 15504 + ESCALA DE LIKERT					


¿Facíl de comprender?
¿Documentación de usuario completa,...
¿Fácil de construir en un sistema compatible?...
¿Fácil de instalar en un sistema compatible?
¿Fácil de aprender a usar sus funciones?
¿La identidad del proyecto / software es clara y única?
¿Es fácil ver quién posee el proyecto / software?
¿Adopción de la licencia apropiada?
¿Fácil de adminsitrar el desarrollo del software?
¿Evidencia de comunidad actual / futura?
¿Evidencia de capacidad de descarga actual / futura?
¿Fácil de probar la corrección del funciones...
¿Utilizable en múltiples plataformas?
¿Evidencia de soporte para desarrolladores actuales /...
¿Fácil de entender a nivel fuente?
¿Fácil de modificar y aportar cambios a los...
¿Evidencia de desarrollo actual / futuro?
¿Interoperable con otro software requerido /...
5
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
Series1
Series2
Series3
Series4
Series5